Lab 5 – Sorting with Linked Lists

Introduction

This lab introduces the concept of pointers, both to data and functions, through the concept of the doubly-linked list data type. This is a very common data type that is used in many programs for storing expandable lists of data. This is another library lab and so the code you add to main() will be solely for your testing purposes.

Reading

- K&R chapter 5, appendix B5

Provided files

- sort.c – contains main(), PrintListItem(), and CompareListItem() along with some other setup code. This is only provided for your debugging purposes only as it will not be submitted with your code.
- LinkedList.h – provides #define constants for return values along with the function prototypes that you will be implementing. Comments above the function prototypes describe their functionality. Do not copy this whole file when creating LinkedList.c, just copy the individual function prototypes.

Assignment requirements

- Your program will implement the functions whose prototypes are provided in LinkedList.h. The functions all have appropriate documentation and describe the required functionality.
  - We have provided sort.c so that you may test and debug your linked list implementation.
- Add inline comments to explain your code (about 1 comment every 6 lines of code is about right).
- Add the following to the top of your LinkedList.c file as comments:
  - Your name
  - The names of colleagues who you have collaborated with
- Submit ONLY LinkedList.c via eCommons before the due date (double check that it was accepted, again BEFORE the due date).

Grading
This assignment again consists of 10 points:

- **5 points** – Properly implemented doubly-linked list
- **3 points** – Proper sorting and printing functionality
- **2 points** – Adhering to the formatting guidelines

You will lose points for the following:

- **-10 points**: code doesn’t compile (seriously)
- **-2 points**: any compiler warnings
- **-2 points**: the files you submit aren’t named as described in this document or you submit more than just the required documents
- **-2 points**: Not using the #define constants for return values
- **-2 points**: if gotos were used
- **-2 points**: sparse commenting (should be about 1 comment per 5-8 lines)

**Doubly-linked lists**

In computer programs, much as in real life, keeping a list of things can be necessary. Usually the number of items that will be in this list is known ahead of time and so in a computer program this list could be kept in an array. There will be occasions, like when processing user input, when the number of items to be stored in a list is not known ahead of time. This is a problem with C’s statically-allocated arrays. The common solution is to use another data type called a linked list.

Linked lists are exactly what they sound like: a collection of objects that are all linked together to form a single list. As each item is linked to at least one other item in the list there is a set ordering to the list: from a “head” item at the start to a “tail” item at the end. Since these items are all connected it is easy to access any item from any other item by just traversing or “walking” through the list.

For this lab you will be implementing a doubly-linked list, the simplest of the linked lists. A doubly-linked list is also straightforward: each item is linked to both the item before it and after it. This allows for traversal of the list from any element to any other element by walking along it, which makes using the list very easy.

The items in the list you are implementing are stored as structs in C because they will be storing a few different pieces of data. Specifically it holds a pointer to the previous ListItem, which will be NULL if it’s the head of the list; a pointer to the next ListItem, which will be NULL if it’s at the end of the list; and a pointer to any kind of data, NULL if there’s no data. The typedef and the name after the “}” let you refer to the struct similar to any other data type, by a single name “ListItem” instead of the longer “struct ListItem”.

The definition of the ListItem struct in LinkedList.h:

```c
typedef struct ListItem {
    struct ListItem *previousItem;
    struct ListItem *nextItem;
```
void *data;
} ListItem;

Now that you understand the structure of a linked list we will introduce the various operations that can be performed upon a list. The standard operations are creating a new list, adding elements to a list, finding the head of a list, and removing elements from a list.

**Creating a new list:** A new list is created by the creation of just single ListItem. Since a list must have a least one element for it to be a list, this new item needs some data to hold, which is generally an argument to the function that creates a new list. As this ListItem is both the head and tail of the list there is no item before it or after it in the list.

**Adding to a list:** Now that you have a new list to use, how do you add more elements to it? With the arrays that you are familiar with, you need to know two things: the position to insert into and the data that will be inserted. With linked lists it's a little different because there's never a “free spot” to insert a new item into. What is done instead is that the position of the new list item is relative to an existing item, generally the item before it in the list. So to insert an item into the list, that item is inserted after an existing item. If the list went A <-> B <-> C and you want to insert D after B then the list would become A <-> B <-> D <-> C. So that means that the previous item and next item pointers of both B and C will need to change to accommodate the new item D.

**Finding the head:** The head of a list is a special item because it has no preceding element (represented by a NULL pointer). Since all the elements in a list are connected, finding the head merely requires traversing the list until a list item is found with no preceding element.

**Removing an element:** Removing an element from a list is the inverse of adding to it. Following the example above you’d go from a list like A <-> B <-> D <-> C to A <-> B <-> C. The pointers of B and C both need to be modified to account for the removal of D. Generally the data that was stored within D is also desired after the removal of the item and should be returned.

**Selection sort**

Sorting is an incredibly important function in computer programming. While you may not think it is used a lot, it is quite common within a program to have the need to sort a series of numbers. Sorting is an entire field of study within computer science and so there is a huge number of algorithms that do just that. Selection sort is the one you will be focusing on in this lab. It is very slow on average, but easy to understand and implement. It is called an in-place sorting algorithm, because it doesn’t need to use a temporary array to store data. Pseudo-code for selection sort is provided below:

```plaintext
for i = 0 to length(A) - 1 do
    for j = i to length(A) - 2 do
            swap A[j] and A[i]
        end if
    end for
end for
```
Selection sort is best understood by thinking of it as building a sorted list on the left by choosing the minimum value from the original unsorted list on the right as the next sorted value. The outer for loop effectively tracks the right-most element of the sorted array filling up the left portion of the array. This means that for every iteration of the outer-loop, the inner-loop can perform many element swaps. An example is shown below. The left-hand column holds the array at the start of an outer-loop iteration with the bold items comprising the already-sorted elements. Each swap within the inner-loop is shown in the right-hand column with the bold items having been swapped.

<table>
<thead>
<tr>
<th>Outer loop array</th>
<th>Inner-loop swaps</th>
</tr>
</thead>
<tbody>
<tr>
<td>64 25 12 22 11</td>
<td>25 64 12 22 11</td>
</tr>
<tr>
<td></td>
<td>12 64 25 22 11</td>
</tr>
<tr>
<td></td>
<td>11 64 25 22 12</td>
</tr>
<tr>
<td>11 25 12 22 64</td>
<td>11 12 25 22 64</td>
</tr>
<tr>
<td>11 12 25 22 64</td>
<td>11 12 22 25 64</td>
</tr>
<tr>
<td>11 12 22 25 64</td>
<td>-</td>
</tr>
<tr>
<td>11 12 22 25 64</td>
<td>-</td>
</tr>
</tbody>
</table>

malloc(), calloc(), and free()

This lab also relies on the use of dynamic memory allocation using malloc(), calloc() and free(). These are touched on in chapter 5 of K&R. As they are standard library functions they are documented thoroughly online or in the Linux man-pages. I refer you to those resources to understand them.